TDP007 - Tentamen

2025-06-12

Regler

- All kod efterrättas på denna tenta
- Inga elektroniska hjälpmedel får medtas. Mobiltelefon ska vara avstängd och ligga i jacka eller väska.
- Inga ytterkläder eller väskor vid skrivplatsen.
- Student får lämna salen tidigast en timme efter start.
- Vid toalettbesök ska vakt informeras.
- All form av kontakt mellan studenter under tentamens gång är strängt förbjuden.
- Böcker och anteckningssidor kan komma att granskas av assistent, vakt eller examinator under tentamens gång.
- Frågor om specifika uppgifter eller om tentamen i stort ska ställas via tentasystemets kommunikationsklient.
- Systemfrågor kan ställas till assistent i sal genom att räcka upp handen.
- Endast uppgifter inskickade före tentamenstidens slut rättas.

Hjälpmedel	En Rubybok (exempelvis the pickaxe) Ett A4-ark med egna anteckningar Tillgång till webresurser: ruby-docs, rubular och tidig version av kursboken

Information

Betygssättning vid tentamen

Tentamen består av 3 delar. Del 1 är uppgifter på grund-nivå som relaterar till tekniker från kursens första två seminarier, dessa bedöms som G/U. Del 2 är uppgifter på grund-nivå som relaterar till tekniker från kursens andra två seminarier, dessa bedöm som G/U. Del 3 består av mer avancerade uppgifter, där varje uppgift relaterar till någon teknik från kursens fyra seminarier. Uppgifter på del 3 ger upp till 10 poäng. Betyg på tentamen ges enligt följande tabell. För alla betyg så måste förutom de insamlade poängen alla uppgifter på Del 1 och Del 2 lösas med godkänt betyg.

Insamlade poäng Del 3	**Betyg**
10	3
17	4
24	5

Duggor

Den första duggan består av två delar. Del 1 är uppgifter på grundnivå som relaterar till tekniker från kursens första två seminarier och bedöms som G/U. Del 2 består av en mer avancerad XML-uppgift som ger upp till 10 poäng. Om alla uppgifter på Del 1 har godkänt betyg från denna dugga så har du Del 1 på ordinarie tentamen tillgodo.

Den andra duggan består av två delar. Del 1 är uppgifter på grundnivå som relaterar till tekniker från kursens sista två seminarier och bedöms som G/U. Del 2 består av en mer avancerad parser-uppgift som ger upp till 10 poäng. Om alla uppgifter på Del 1 har godkänt betyg från denna dugga så har du Del 2 på ordinarie tentamen tillgodo.

Om del 1 är avklarad på båda duggorna och du har samlat in totalt 14 poäng från duggorna så har du förutom Del 1 och Del 2 på ordinarie tentamen tillgodo också betyg 3 på tentamensmomentet.

Inloggning

Logga in på datorn i labbsalen med ditt liu-id och lösenord. Detta är samma inloggningsuppgifter som du använder i Lisam.

Skrivbordsmiljön

När du kommit in i tentasystemet har du en normal skrivbordsmiljö (Mate-session i Ubuntu). Efter en stund kommer även din kommunikationsklient att dyka upp automatiskt. Startmenyn är nedskalad till enbart det som examinator bedömt relevant. Andra program kan fortfarande finnas tillgängliga genom att starta dem från en terminal. Observera att en del program som använder nätverkstjänster inte fungerar normalt, eftersom nätverket inte är åtkomligt.

När du är inloggad är det viktigt att du har tentaklienten igång hela tiden. Om den inte dykt upp fem minuter efter inloggning och inte heller när du startar den manuellt från menyn (fisken) tar du kontakt med assistent eller vakt i sal.

Terminalkommandon

`ruby` används för att köra en ruby-fil.
`irb` används för att starta ruby i interaktivt läge.
`ri` används för att komma åt den inbyggda dokumentationen.

Ruby-docs

På tentan har du tillgång till referenssidorna på `https://ruby-doc.org/` (både core och std-lib finns tillgänglig).

Rubular

På tentan har du tillgång till sidan `https://rubular.com/`.

Givna filer

Eventuella givna filer finns i katalogen `given_files`. Denna underkatalog är skrivskyddad, så det är ingen risk du råkar ändra på dessa filer. Skrivskyddet gör dock att du måste kopiera in de givna filer du vill använda till tentakontots hemkatalog. Hur du listar och kopierar filer ska du kunna. Hemkatalogen står du i från början, och du kommer alltid tillbaka till den genom att bara exekvera kommandot `cd` i terminalen.

Avslutning

När du skickat in alla uppgifter och känner dig färdig kan du logga ut och lämna salen. Antalet poäng meddelas i efterhand.

Avsluta alla öppna program och logga ut ur datorn. Lämna inte datorn innan du ser att du är utloggad.

Del 1: Uppgift 1 - SAX-parser

I filen `games.xml` finns data för spel i Pontus samling, det finns här krigsspel (wargames) och rollspel (roleplayingGames). Varje spelsystem har ett namn och en komplexitet. Ditt jobb är att med en Strömparser (SAX) hämta ut en lista över alla namn på spel (krigsspel och rollspel) som har låg komplexitet. Listan av namn ska komma ut från parsern till huvudprogrammet och skrivas ut på något sätt (se funktionella krav nedan). Exakt formattering av utskriften spelar ingen roll, men listan behöver innehålla korrekt data.

Funktionella krav:

```
["Frostgrave", "Gaslands", "FATE Core", "MÖRK BORG", "Cairn"]
```

Del 1: Uppgift 2 - DOM-parser

Lös samma problem som i Uppgift 1, men använd nu en DOM-parser istället för en SAX-parser. Använd minst ett xpath-uttryck.

Funktionella krav:

```
["Frostgrave", "Gaslands", "FATE Core", "MÖRK BORG", "Cairn"]
```

Del 2: Uppgift 3 - DSL

I filen `dsl.rb` finns en påbörjad representation av Pontus samling av spel. I den givna filen `uppgift3.rb` finns en påbörjad läsare för detta DSL. Du ska modifiera denna läsare så att den med `method_missing` läser ut information från filen som sedan skrivs ut enligt de funktionella kraven. `game_system` kan hanteras med en separat funktion, men resterande data om varje spel ska hanteras med `method_missing` och ska vara generell nog för att hantera att fler spel och fler data läggs till om varje spel. Du kan anta att datan alltid är strängar eller heltal som i exempelfilen. Formatteringen av utskriften spelar ingen roll i denna uppgift, inte heller datatyperna, det behöver bara vara läsbart och innehålla rätt information.

Funktionella krav:

```
{
  "Dungeons & Dragons 5e"=>
    {:genre=>["Fantasy"]},
  "Call of Cthulhu"=>
    {:genre=>["Horror"], :complexity=>["Medium"]},
  "Ironsworn"=>
    {:genre=>["Dark Fantasy"], :complexity=>["Medium"], :players=>[1, 3]},
  "Thousand Year Old Vampire"=>
    {
      :genre=>["Solo Narrative"],
      :complexity=>["Low"],
      :players=>[1],
      :mechanics=>["Journal", "Prompts"]
    }
}
```

Del 2: Uppgift 4 - Parser

I `uppgift4.rb` finns en påbörjad aritmetikparser. Utöka denna parser så att den kan hantera att variabler tilldelas och läses. Godkända variabelnamn är enskilda bokstäver, a till och med z. Din lösning ska fungera med det givna huvudprogrammet. Utskriften behöver inte formateras som i de funktionella kraven, men testerna som framgår där ska finnas med i din lösning, med utskrifter som gör att vi kan se att det fungerar som förväntat.

Funktionella krav:

```
a = 3 >>> 3
a >>> 3
b = a * 2 >>> 6
a >>> 3
b >>> 6
a = b / (a - 1) >>> 3
```

Del 3: Uppgift 5 - Spara uttryck för framtida körning (10p)

I filen uppgift5.rb hittar du en parser som påminner om det språk (*DiceRoller*) vi arbetat med under föreläsningarna. Mindre ändringar har gjorts så att parsern nu körs med inskickade strängar istället för inmatningar i terminalen och att den bara hantera grundläggande aritmetik.

(1) Skriv om parsern så att denna returnerar ett Abstrakt Syntax Träd (AST) istället för att utvärderas direkt. Hur du representerar detta AST är upp till dig, men vi rekommenderar att du använder instanser av klasser för att representera noderna. Ditt AST ska skapas för varje rad som läses av parsern och sedan utvärderas direkt. Lämpligt är att justera medlemsfunktionen `eval` i klassen `AritLang` så att trädet som nu returneras först utvärderas och att det är resultatet av utvärderingen som skrivs ut.

(2) Lägg till hantering av exponenter med korrekt prioritet och associativitet.

(3) Justera den givna koden så att användaren kan deklarera och anropa förenklade funktioner. En förenklad funktion låter användaren spara undan ett godtyckligt uttryck som senare kan köras. Dessa förenklade funktioner kan bara spara ett aritmetiskt uttryck, hanterar inga parametrar eller scope. Programmet behöver bara kunna lagra en funktion åt gången. Det är alltså alltid den senast definierade funktionen som körs vid anrop. Senaste funktionen sparas med fördel som global variabel i denna uppgift, globala variabler arbetar man med på samma sätt som datamedlemmar men man använder dollartecken istället för snabel-a `$varname`.

Funktionella krav:
Returvärdet vid körning av `define` spelar ingen roll i uppgiften, men övriga delar av funktionella kraven bör stämma med körning av ditt program.

```
1 + 2 >>> 3
2 ^ 2 ^ 3 >>> 256
1 + (1 + 2) * 3 ^ 2 >>> 28
define 1 + 2 >>> #<Arithmetic:0x000055dde23922e8>
call >>> 3
define 2 ^ 2 ^ 3 >>> #<Arithmetic:0x000055dde23479f0>
call >>> 256
define 1 + (1 + 2) * 3 ^ 2 >>> #<Arithmetic:0x000055dde23da778>
call >>> 28
```

Del 3: Uppgift 6 - Icke-deterministisk programmering (10p)

I filen `uppgift6.rb` finner du den AMB-klass vi arbetat med under föreläsningsserien, kom ihåg att det saknas implementation av medlemsfunktionen `next`. Din uppgift är att använda denna för att lösa följade problem.

Du har anlitats för att hjälpa till att skapa en schemaläggare för ett lokalt café. På detta café jobbar det fyra personer: Anna, Bosse, Cajsa och Daniel. Cafét har fyra arbetspass per dag, ett på morgonen, två på eftermiddagen och ett på kvällen. Varje anställd får schemaläggas på som mest ett skift per dag.

Det finns dock lite problem att ta hänsyn till här. Bosse kan inte jobba på några kvällspass, och det är bara Anna och Cajsa som kan jobba på morgonpassen.

Med hjälp av den icke-deterministiska problemlösaren, ändra i `my_function` så att alla möjliga schemaläggningskombinationer skrivs ut. Hur denna utskriften ser ut spelar ingen roll, det viktiga är att informationen är korrekt och att det framgår tydligt vilken person som jobbar på vilket pass. Det funktionella kravet visar inte alla kombinationer, utan endast en delmängd för att visa ett exempel.

Funktionella krav:

```
Föreslaget schema:
  Morgon: Anna
  Eftermiddag 1: Bosse
  Eftermiddag 2: Cajsa
  Kväll: Daniel
--------------------
Föreslaget schema:
  Morgon: Anna
  Eftermiddag 1: Bosse
  Eftermiddag 2: Daniel
  Kväll: Cajsa
--------------------
...
--------------------
Föreslaget schema:
  Morgon: Cajsa
  Eftermiddag 1: Bosse
  Eftermiddag 2: Daniel
  Kväll: Anna
--------------------
Föreslaget schema:
  Morgon: Cajsa
  Eftermiddag 1: Daniel
  Eftermiddag 2: Bosse
  Kväll: Anna
--------------------
Det var alla möjligheter. 8 möjliga lösningar hittades.
```

Del 3: Uppgift 7 - XML-parsning och rekursion (10p)

Love spelar för närvarande alldeles för mycket Satisfactory, och har bett om din hjälp för att skriva ett verktyg som ska hjälpa honom holla koll på sina produktionskedjor. Detta är ett spel som går ut på att bygga fabriker som tillverkar olika produkter och varje produkt kan därefter användas för att bygga andra produkter. För detta ändamål har du fått två filer: `satisfactory.xml` och `uppgift7.rb`. I `uppgift7.rb` finns ett påbörjat huvudprogram som du kan utgå ifrån.

I filen `satisfactory.xml` finner du alla ingredienser och delingredienser som spelet kräver för att kunna skapa **Heavy Modular Frames**. För detta krävs det en massa olika ingredienser, men allting skapas av tre grundläggande beståndsdelar som ligger i elementet `base_resources`. I elementet `recipes` hittar du alla recept som behövs för att skapa Heavy Modular Frames. Varje recept berättar hur mycket av en viss produkt som behövs (***`demand`***) för att skapa en upplaga av produkten. Ett recept kan behöva antingen produkter som skapats i andra recept, eller någon av våra `base_resources`.

Du ska använda en XML-parser (*en DOM-parser är att rekommendera, men du får göra det med en SAX-parser om du vill*) som parsar in alla basresurser. Sedan ska du skapa en funktion där användaren anger ett namn på en produkt de vill skapa. Programmet ska sedan, med hjälp av XML-parsern, berätta hur många resurser och eventuella andra produkter som krävs för att skapa den givna produkten. Även om det inte är ett strikt krav så är det smidigast att göra beräkningen av resurserna rekursivt.

För att göra en `Steel Beam` krävs exempelvis: 4 stycken `Steel Ingot`. Dessa behöver i sin tur 1 stycken `Iron Ore` och 1 stycken `Coal` vardera. Eftersom både `Iron Ore` och `Coal` är basresurser så är vi klara där. Vi kan då svara att vi behöver 4st `Steel Ingot`, 4st `Iron Ore` och 4st `Coal` för att bygga en `Steel Beam`

Funktionella krav:
Vänstra exemplet visar åtgången för att bygga en `Heavy Modular Frame` och högra exemplet visar åtgången för att bygga en `Encased Industrial Beam`. Din utskrift behöver vara läsbar och innehålla korrekt data, men formateringen spelar i övrigt ingen roll.

```
[["Coal", 90.0],
 ["Concrete", 30.0],
 ["Encased Industrial Beam", 5.0],
 ["Iron Ingot", 150.0],
 ["Iron Ore", 240.0],
 ["Iron Plate", 45.0],
 ["Iron Rod", 82.5],
 ["Limestone", 90.0],
 ["Modular Frame", 5.0],
 ["Reinforced Iron Plate", 7.5],
 ["Screw", 210.0],
 ["Steel Beam", 15.0],
 ["Steel Ingot", 90.0],
 ["Steel Pipe", 20.0]]
```

```
[["Coal", 12.0],
 ["Concrete", 6.0],
 ["Iron Ore", 12.0],
 ["Limestone", 18.0],
 ["Steel Beam", 3.0],
 ["Steel Ingot", 12.0]]
```